**CS506 Programming for Computing**

**HOS10 Text Generation Using LSTM**

2/22/2025 Developed by Jin Chang

5/19/2025 Reviewed by Clark Ngo

05/30/2025 Reviewed by Naveena Moddu

05/30/2025 Reviewed by Ashwin Paturi

School of Technology & Computing (STC)

City University of Seattle (CityU)



**Before You Start**

* **Screenshots may be different from your environment.**
* The directory path shown in screenshots may be different from yours.
* There might be subtle discrepancies along with the steps. Please use your best judgment while going through this cookbook-style tutorial to complete each step.
* Some steps may not be explained in detail. If you are not sure what to do:

1. Consult the resources from the course.
2. If you cannot solve the problem after a few tries (usually 15 -30 minutes), ask a TA for help.

**Learning Outcomes**

Students will be able to:

* Understand the Long Short-Term Memory network building, Training and Testing.
* Understand how to perform text automatic generation using LSTM.
* Able to demonstrate in Python using Keras modules to create, train and test text automatic generation using LSTM.

**Resources**

* Text resource (for training): <https://www.gutenberg.org/ebooks/75469>

**Section 1: Preparing your environment -** Get started with your virtual environment here: <https://cityuseattle.github.io/docs/git/github_codepsace/#codespaces>

**Section 2: Text Generation using LSTM**

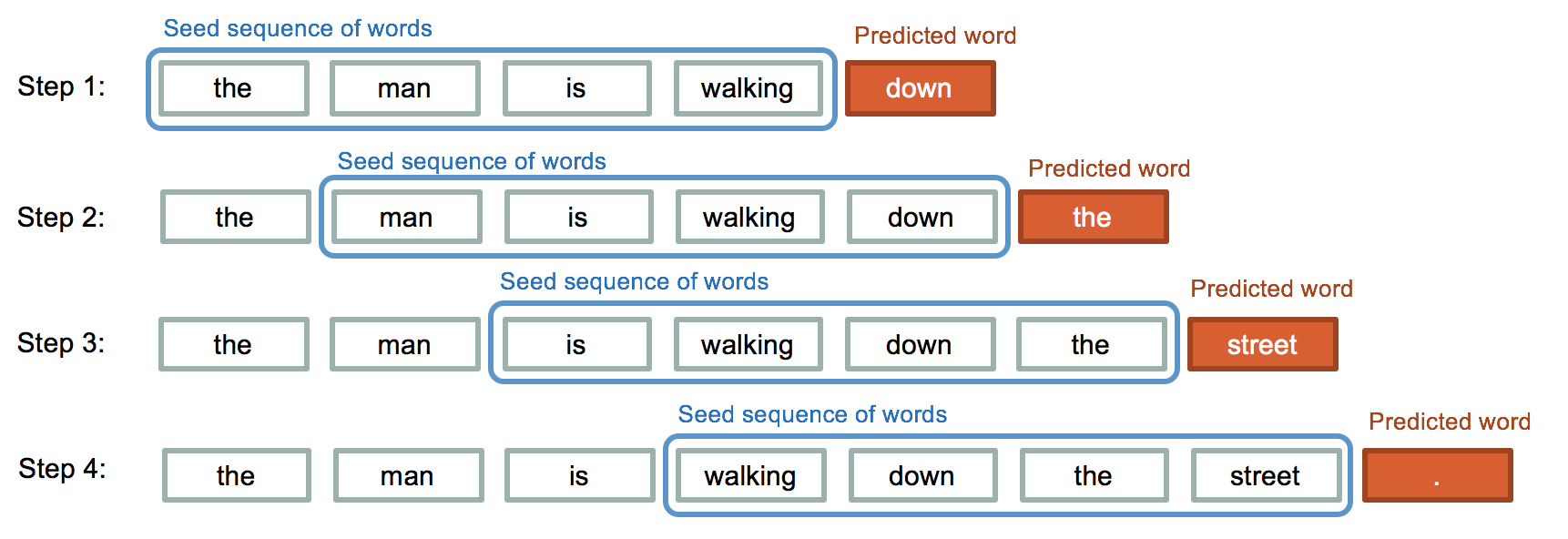
**Introduction to LSTM**

Long short-term memory (LSTM) units (or blocks) are a building unit for layers of a recurrent neural network (RNN). A RNN composed of LSTM units is often called an LSTM network. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell is responsible for "remembering" values over arbitrary time intervals; hence the word "memory" in LSTM. Each of the three gates can be thought of as a "conventional" artificial neuron, as in a multi-layer (or feedforward) neural network: that is, they compute an activation (using an activation function) of a weighted sum. Intuitively, they can be thought of as regulators of the flow of values that goes through the connections of the LSTM; hence the denotation "gate". There are connections between these gates and the cell.

The expression long short-term refers to the fact that LSTM is a model for the short-term memory which can last for a long period of time. An LSTM is well-suited to classify, process and predict time series given time lags of unknown size and duration between important events. LSTMs were developed to deal with the exploding and vanishing gradient problem when training traditional RNNs.

**Automatic Text Generation Using LSTM**

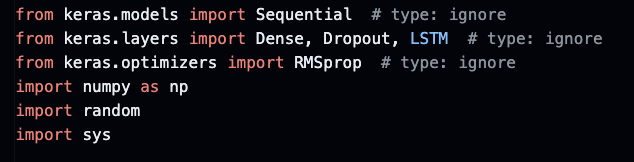
Automatic Text Generation using LSTM is a method of generating text automatically using a type of neural network called a Long Short-Term Memory (LSTM), which is particularly adept at handling sequential data like language allowing it to generate coherent sentences or paragraphs by predicting the next word based on the previous context within the sequence. Here is how it works: The model is trained on large amounts of text data, learning the probability of different words appearing next to each other. When generating text, the LSTM takes an initial word or phrase as input and then predicts the following word based on its learned patterns, iteratively building a complete sentence or paragraph. As shown below, given 4 sequences of words (or letters), it can predict the next word (or a letter), slide down 1 or more depending on the size of training data, feed the next sequence to the network and predict the next word based on the training.



(source: <https://kgptalkie.com/text-generation-using-tensorflow-keras-and-lstm/>)

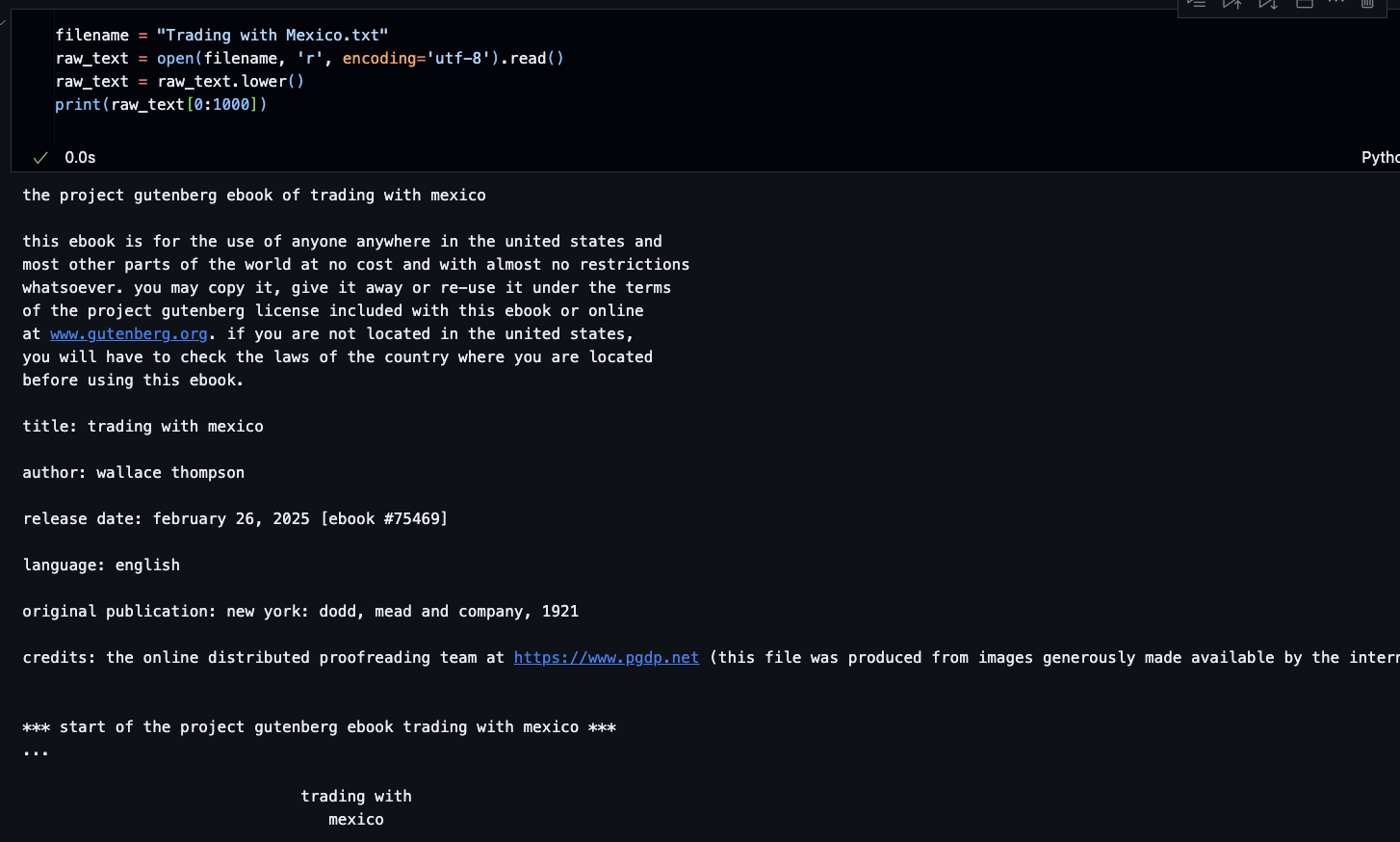
**Implementation of Automatic Text Generation Using LSTM**

1. Let’s import the necessary modules to use LSTM.

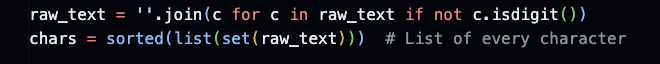


1. Load the text a book titled Trading with Mexico.txt (from www.gutenberg.org) with encoding utf-8. If you like to use a different text, then make sure you open the text using text editor and save it to UTF-8 encoding as shown below.

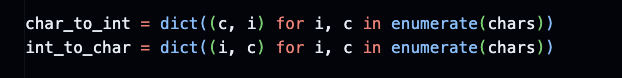
Once you read the file and lower case the raw texts and you can print the first 1000 characters to make sure you loaded the correct texts for your training.



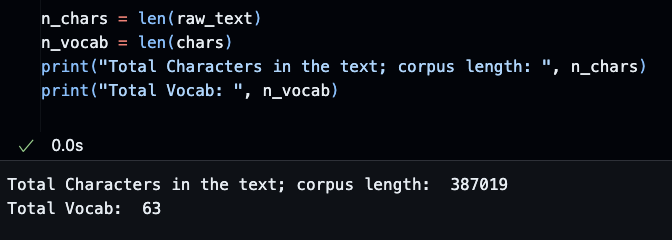
1. Let’s clean up these texts by removing the digits in the file. We can store those ‘unique’ characters, sort and store them into chars.



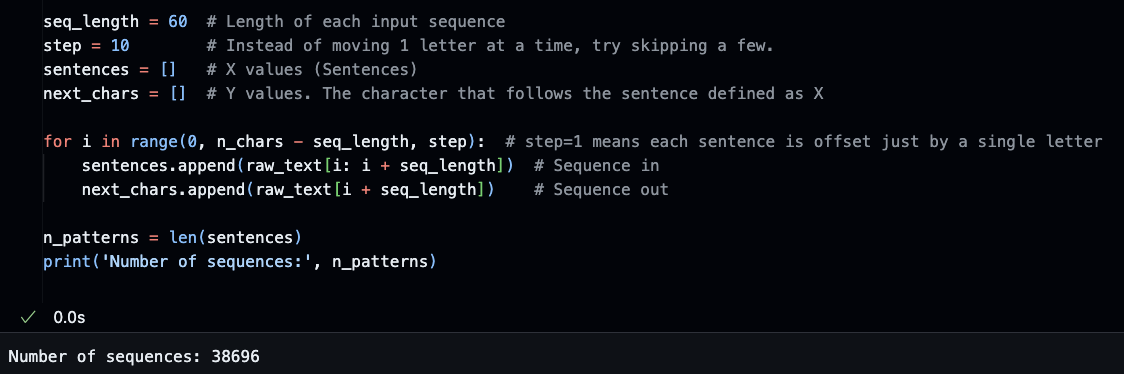
1. Since all the character sequence should be encoded as integers, each unique character needs to be assigned to an integer value. Also we can reverse this so that we reverse engineer our predictions in characters rather than integer values.



1. Let’s review what we have here for total characters in the text (= corpus length), and total number of unique characters which is going to be our vocabulary. In this case, we have 63 unique characters, and this number includes white spaces or other punctuations as well.



1. So far, we have the characters we created input and output sequences for training and note that LSTM input and output can be sequences. X values are sentences as input, and Y values are the output character to follow based on the sequence found in X. For example, “I study at the librar” as an input sequence, the output character would be ‘y’ which will be stored in next\_chars. We’ll set the sequence to be 60 and skip 10 characters.

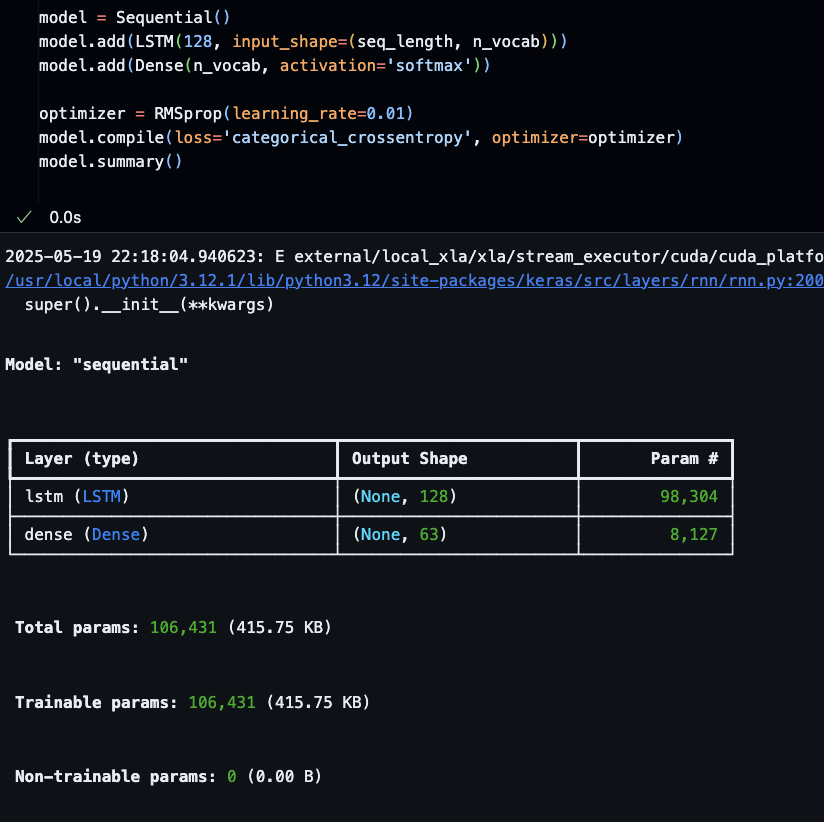


1. Remember the figure above, x is the sequence (input) and y would be the input (sequence) plus next\_chars that comes after the sentence. Those x and y are very similar to the one-hot encoding in Chapter 6. In this case, one-hot encoding of the labels consists of embedding each label as an all-zero vector with a True in the place of the label index otherwise lots of False values. In detail, first, initializes the input tensor with shape (number of sentences, sequence length, vocabulary size). Then initializes the target tensor with shape (number of sentences, vocabulary size). Now it loops over each sentence in the dataset as iterating over each character in the current sentence. It maps the current character to its integer index in the vocabulary if t and char pair is found. And y gets True when the target at time step t to 1 for the next character in the sequence. This vectorization prepares the data for LSTM training by creating one-hot encoded inputs and targets for each sentence in the dataset. Vectorization returns a vector for all sentences indicating the presence or absence of a character.



1. In the next step, we are now going to build the model. This LSTM network model is designed for sequence-to-sequence learning tasks, such as language translation or text generation. Here's a breakdown of its components:
   1. Input layer: The LSTM expects input sequences of varying lengths, so it uses a sequence length parameter and a vocabulary size (n\_vocab).
   2. LTSM Layer: This is the core of the network, with 128 units. It processes the input sequences, learning to capture long-term dependencies in the data.
   3. Dense Layer: This final layer has the same vocabulary size as the input layer. It uses a softmax activation function to produce probability distributions over the vocabulary for each time step.
   4. Optimizer: The RMSprop optimizer is used, with a learning rate of 0.01.
   5. Loss Function: Categorical crossentropy is chosen as the loss function, which is suitable for multi-class classification tasks.
   6. Summary: The model.summary() function is called to provide an overview of the model's architecture.

This LSTM model is set up to learn from sequences of varying lengths and produce output sequences of the same vocabulary size as the input. The vectorization process prepares the data for training, allowing LSTM to learn patterns and map input sequences to output sequences.



1. The checkpoint shown is a model checkpoint object used in Keras for saving and loading trained models. Here's a breakdown of its components:

1. \*\*\_\_init\_\_(filepath, monitor, verbose, save\_best\_only, mode)\*\*:

- \*\*filepath\*\*: The path where the checkpoint will be saved.

- \*\*monitor\*\*: The metric to track for saving the checkpoint (in this case, 'loss').

- \*\*verbose\*\*: Level of verbosity (1 for progress messages).

- \*\*save\_best\_only\*\*: Whether to only save the best checkpoint (True in this case).

- \*\*mode\*\*: The comparison mode (min in this case, meaning it will save the checkpoint if the monitor value improves).

2. \*\*filepath\*\*: "saved\_weights(epoch:02d)(loss:4f).keras" is the path where the checkpoint will be saved. It includes the epoch number (02d) and the loss value (4f) in the filename.

3. \*\*call(model)\*\*: This method is called during training to save the weights of the model if the monitor value improves.

This checkpoint setup allows the model to save its weights periodically during training, ensuring that the best weights can be restored later if needed. It's particularly useful for long training processes or when training on large datasets where saving the entire model after every epoch would be impractical.



1. We can now fit the model. The model.fit() function shown is used to train the LSTM model on the prepared data. Here's a breakdown of the key components: 1. \*\*model\*\*: The LSTM model instance to be trained.

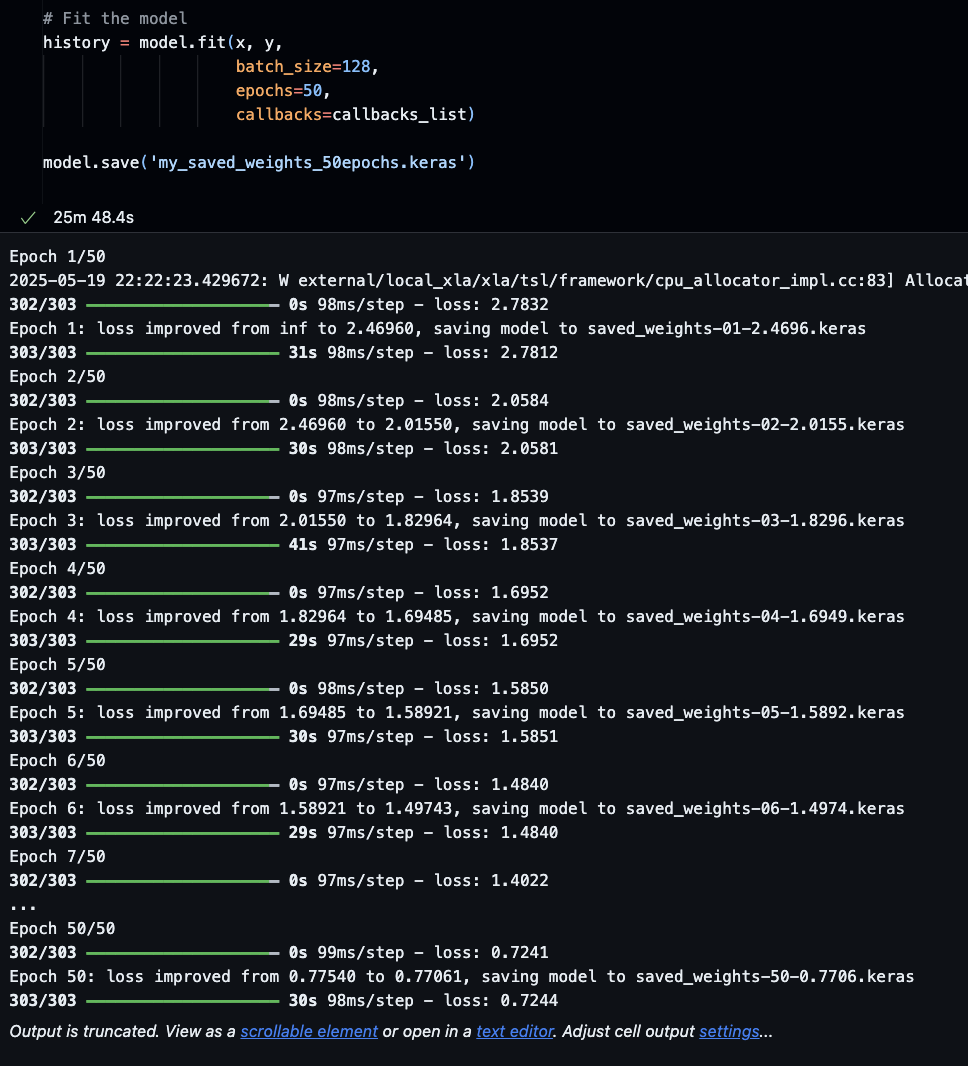
2. \*\*X, y\*\*: The training data, where X is the input sequences and y is the target sequences.

3. \*\*batch\_size\*\*: 128, the number of samples per gradient step.

4. \*\*epochs\*\*: 50, the number of training epochs.

5. \*\*callbacks\*\*: A list of callbacks to be called during training. In this case, only one callback is provided: the model checkpoint.

The model.fit() function will train the LSTM on the input data for 50 epochs, using a batch size of 128. After each epoch, it will call the checkpoint callback to save the best weights if the monitor value (likely the loss) improves. This training process allows the LSTM to learn patterns in the input sequences and map them to the output sequences effectively, using the vectorized data prepared earlier in the code.



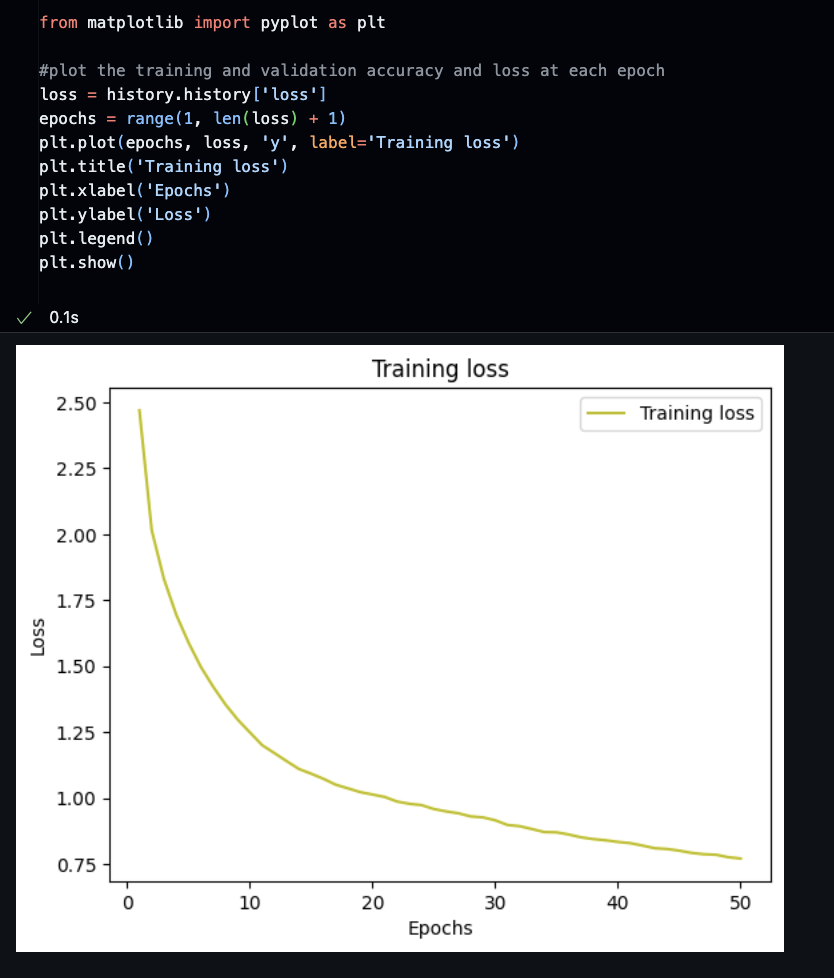
1. The code snippet shows a loop that prints the loss at each epoch iteration during the training process. Here's a breakdown:

1. \*\*iteration loop\*\*: This loop iterates over each epoch, running from 1 to 50 (the number of epochs specified in the model.fit() call).

2. \*\*epoch loop\*\*: For each epoch, it loops over the training data (X, y).

3. \*\*training loop\*\*: Inside each epoch, it loops over the training data in batches. 4. \*\*loss printing\*\*: After each batch, the current loss is printed to the console.

This code provides a way to monitor the training progress visually by observing the loss decrease over time. It allows developers to track the model's performance as it learns, ensuring that the training is proceeding as expected and that the model is improving with each epoch.



1. The LSTM sample method shown in the code snippet is used to generate new sequences based on the learned model. Here's a breakdown:

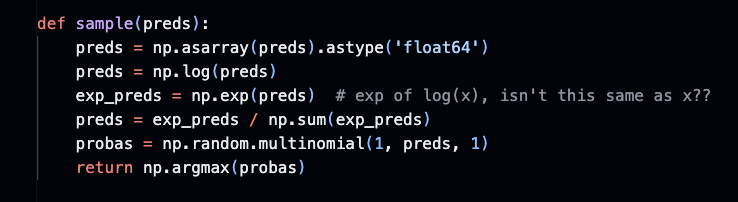
1. \*\*initialize state\*\*: It initializes the LSTM state with zeros, which is necessary to start the generation process.

2. \*\*generate sequence\*\*: It then iterates over a fixed sequence length (30 in this case).

3. \*\*predict next character\*\*: For each time step, it uses the current state and input (which is likely the last predicted character) to predict the next character.

4. \*\*update state\*\*: Based on the predicted character, it updates the LSTM state.

This method demonstrates how an LSTM trained on sequences can be used to generate new sequences, which is a common application in tasks like text generation or sequence prediction. The fixed sequence length suggests that this is likely a demonstration of the model's ability to generate a fixed-length output sequence, rather than an end-to-end sequence-to-sequence translation. We must provide a sequence of seq\_length as input to start the generation process. The prediction results is probabilities for each of the 48 characters at a specific point in sequence. Let us pick the one with max probability and print it out.



1. The prediction step in an LSTM execution, as shown in the code, involves several key components:

1. \*\*initialize state\*\*: The LSTM hidden state and cell state are initialized, typically with zeros. This is crucial for the LSTM to start processing the input sequence.

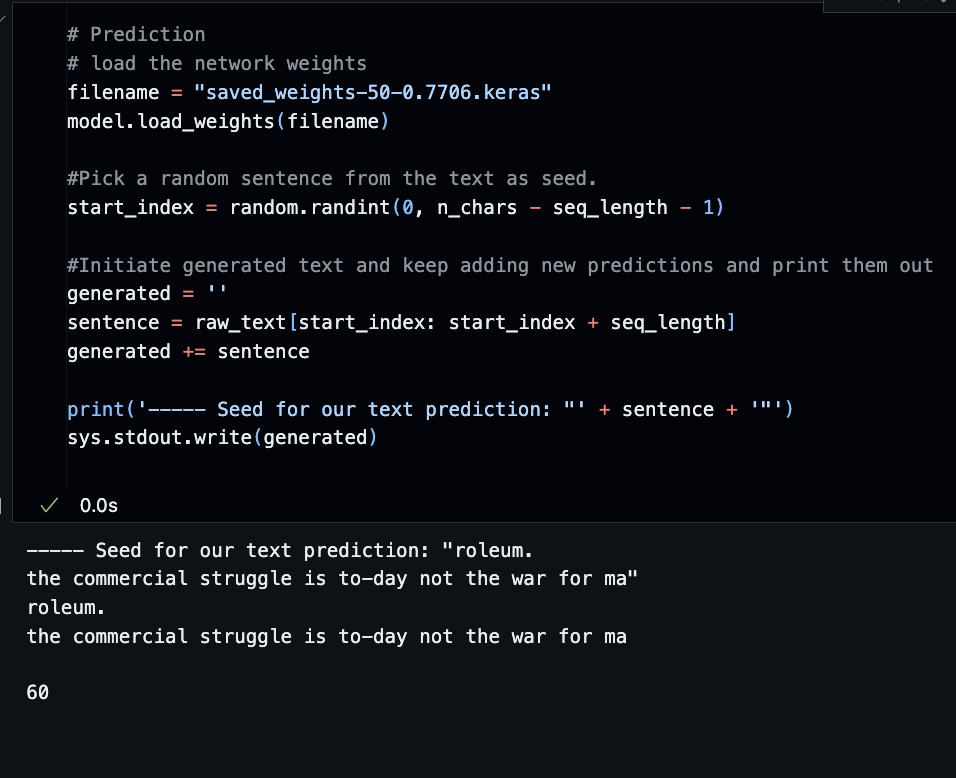
2. \*\*input processing\*\*: The first input to the LSTM is often a special start token or the initial character of the sequence to be generated.

3. \*\*iteration\*\*: The LSTM then iterates over the input sequence, one time step at a time.

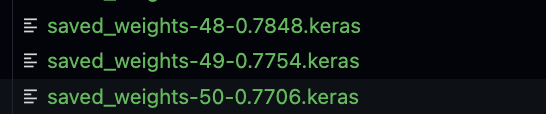
4. \*\*output and state update\*\*: For each time step, the LSTM computes the output (which is typically the next character in the sequence) and updates its hidden state based on the input and current state.

5. \*\*state carry-over\*\*: The updated hidden state from one time step is carried over to the next, allowing the LSTM to maintain context from earlier parts of the sequence.

This prediction step is how the trained LSTM generates new sequences. It takes the current state and input as input, produces an output (the next character), and updates its state. This process is repeated for the desired number of time steps to generate a fixed-length sequence.



**Note: The “filename” might differ. Please refer to your own generated file. Example:**



1. The text automatic generation function shown in the code snippet is a simple yet effective way to generate new text based on a pretrained LSTM model. Here's a breakdown:

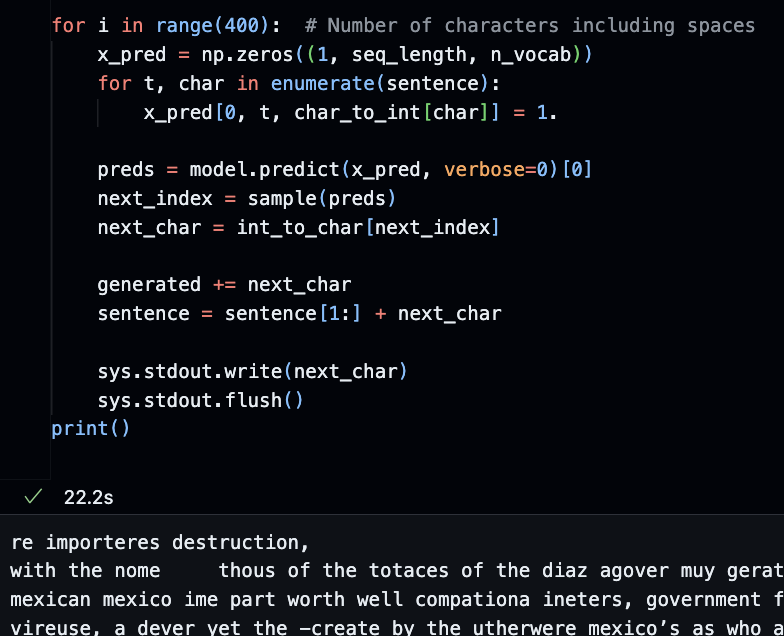
1. \*\*initialize state\*\*: It initializes the LSTM state with zeros, starting the generation process.

2. \*\*generate sequence\*\*: It then iterates over a fixed sequence length (30 in this case).

3. \*\*predict next character\*\*: For each time step, it uses the current state and input (which is likely the last predicted character) to predict the next character.

4. \*\*update state\*\*: Based on the predicted character, it updates the LSTM state.

This function demonstrates how an LSTM trained on sequences can be used to generate new sequences, which is a common application in tasks like text generation or sequence prediction. The fixed sequence length suggests that this is likely a demonstration of the model's ability to generate a fixed-length output sequence, rather than an end-to-end sequence-to-sequence translation. It's a straightforward way to showcase the LSTM's language modeling capabilities. Of course, there are many wrong words too because of the quality of model and training.



**Section 3: Pushing your work to GitHub**

Follow instructions here: <https://cityuseattle.github.io/docs/git/codespaces_submission/>

1. Go to Source Control on your GitHub Codespaces and observe the pending changes.
2. Type the message for your changes in the message box at the top.
   * For example,” **Submission for Module10 – Your Name**”
3. Click on the dropdown beside the commit button and select “**Commit & Push”** to update the changes to your repository main branch.
   * Select **Yes** when prompted.